Parallelization and analysis of selected numerical algorithms using OpenMP and Pluto on symmetric multiprocessing machine

Received 8 May 2018
Revised 8 November 2018
Accepted 16 November 2018

Tanvir Habib Sardar
School of Engineering and Technology, Jain University, Bengaluru, India, and
Ahmed Rimaz Faizabadi
P.A. College of Engineering, Mangalore, India

Abstract

Purpose – In recent years, there is a gradual shift from sequential computing to parallel computing. Nowadays, nearly all computers are of multicore processors. To exploit the available cores, parallel computing becomes necessary. It increases speed by processing huge amount of data in real time. The purpose of this paper is to parallelize a set of well-known programs using different techniques to determine best way to parallelize a program experimented.
Design/methodology/approach – A set of numeric algorithms are parallelized using hand parallelization using OpenMP and auto parallelization using Pluto tool.
Findings – The work discovers that few of the algorithms are well suited in auto parallelization using Pluto tool but many of the algorithms execute more efficiently using OpenMP hand parallelization.
Originality/value – The work provides an original work on parallelization using OpenMP programming paradigm and Pluto tool.

Keywords Algorithms, OpenMP, Auto parallelization, Code parallelization, Hand parallelization, Pluto
Paper type Research paper

1. Introduction

Parallelization is the act of designing a computer program to process data in parallel. Without parallelization, computer programs compute data serially, they solve one problem and then move to the next. If a computer program is parallelized, it breaks a problem down into smaller pieces that can each independently be solved at the same time by discrete computing resources (Thakkar *et al.*, 2017). The parallel machines are being built to satisfy the increasing demand of higher performance for numerous applications. Multi and many-core architectures are becoming a hot topic in the fields of computer architecture and high-performance computing (Culler *et al.*, 1999). The availability of multicore processor has made the programmer to change their way of computing. While creating a new application, the programmer prefers parallel computing to obtain the efficient performance in valuable time and to utilize the available cores. There is a need to either re-write them or convert them to parallel using some tools.

There are different automatic parallel tools built depending on the hardware architecture, memory architecture, data and control dependencies in the software. Each tool differs from other based on the application they parallelize. These tools reduce the manual analysis burden, time and effort (Athavale *et al.*, 2011). Automatic parallelization of sequential programs consists of two components: extraction of parallelism and generation of parallel code for the target architecture. Techniques have been developed to transform sequential code and extract parallelism out of it automatically (Thouti and Sathe, 2013), which can be used to generate parallel code for any parallel architecture. The parallel code

Data Technologies and Applications
© Emerald Publishing Limited
2514-9288
DOI 10.1108/DTA-05-2018-0040

provides efficiency in code execution by dividing the job into many threads or processor (Parallel Processing & Parallel Databases, 2016).

1.1 Automatic parallelization

Automatic parallelization aims at extracting the parallelism out of a source program. Automatic parallelization distributes sequential code into multi-threaded code. It automatically generates parallel (multi-threaded) code for specific loop constructs. The goal of automatic parallelization is to help the programmers to reduce the burden from the error-prone manual parallelization process. In general, automatic parallelization focuses on loops in the program because most of the execution time of a program takes place inside some form of loop. Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation (Jindal *et al.*, 2012).

1.2 Need for automatic parallelization

Writing parallel programs is strictly more difficult than writing sequential program. In sequential programming, the programmer must design an algorithm and then express it to the computer in some manner that is correct, clear and efficient to execute. Parallel programming involves these same issues, but also adds a number of additional challenges that complicate development. These challenges include: finding and expressing concurrency, managing data distributions, managing inter-processor communication, balancing the computational load and implementing the parallel algorithm correctly.

1.3 Numeric problems (algorithms)

Problems can be classified into various types such as numeric, non-numeric, affine, non-affine, dense, sparse, regular and irregular. Programs written to solve mathematical problems and whose outputs are expressed by numbers, instead of letters, are called numeric programs. To investigate physical problems, we develop mathematical models and then attempt to solve the equations. Often this cannot be done analytically and instead we must use a computer. But in the case of using a computer program to solve a mathematical problem, first we must develop an algorithm, a set of instructions, the computer can follow to perform the numerical task we are interested in. Next we need to develop a program to implement the algorithm. Importantly, we must then test our program, and finally we can run the program to generate the results we need.

Solving mathematical problems using equations with integer or floating point values such problems are matrix multiplication, matrix addition, Runge–Kutta method, Monte Carlo method, Lagrange's inverse interpolation method, polynomial addition, subtraction and multiplication, fast Fourier transform and many more (Soleymani, 2012; Gopan *et al.*, 2005). Considering the matrix multiplication, it is a binary operation that takes a pair of matrices, and produces another matrix. Numbers such as real and complex numbers can be multiplied according the elementary arithmetic. The sorting of the elements in numerical order is another form of numeric programs such as Quicksort, Merge sort, Insertion sort, Bubble sort (Kulalvaimozhi *et al.*, 2015; Al-Kharabsheh *et al.*, 2013; Gupta, 2016). Finding the shortest path algorithm also falls under the numeric category. Examples are Dijkstra's algorithm, Bellman–Ford algorithm and Floyd–Warshall algorithm.

1.4 Shared memory architecture

In the shared memory architecture, the entire memory, i.e., main memory and disks, is shared by all processors. A special, fast interconnection network (e.g. a high-speed bus or a cross-bar switch) allows any processor to access any part of the memory in parallel. All processors are under the control of a single operating system which makes it easy to deal with load balancing. It is also very efficient since processors can communicate via the main memory. Figure 1 represents shared memory architecture.

The objective of this work is to parallelize the classified problems and then analyze them on shared memory architecture and to establish the ease for migration of application on shared memory architecture using hand parallelization and auto parallelization. The scope is to help the programmer to understand the benefits of auto parallelization over hand parallelization for classified problems on shared memory architecture.

2. Literature survey

2.1 Automatic parallelization tools

There are many automatic parallelization tools that are reported in the literature. A few important tools are described here. Autopar (Kalender *et al.*, 2014) achieves automatic parallelization of recursive applications using static program analysis. It first selects on the recursive parts of a given program. Then, it analyzes and collects information about these recursive functions. Finally, it achieves automatic parallelization by introducing necessary OpenMP pragmas in appropriate places in the application. Bones (Nugteren and Corporaal, 2012) is a source-to-source compiler based on algorithmic skeletons and a new algorithm classification. The compiler takes C code annotated with class information as input and generates parallelized target code. Cetus (Dave *et al.*, 2009) emphasizes on automatic parallelization targeting C programs. It supports source-to-source transformations, is user oriented and easy to handle and provides the most important parallelization passes as well as the underlying enabling techniques. The infrastructure project follows Polaris (2016), which is a research infrastructure for optimizing compilers on parallel machines. While Polaris translated FORTRAN, Cetus targets C programs. Low-level virtual machine (Lattner and Adve, 2004) is a compiler framework designed to support program analysis and transformation for arbitrary programs by providing high-level information to compiler transformation at compile time, link time, runtime and in idle time between runs. By defining low-level code representation in Static Single Assignment form, it can provide a language-independent representation. Pluto (Pluto – Introduction, 2016) is an automatic parallelization tool based on the polyhedral model. The polyhedral model for compiler optimization provides an abstraction to perform high-level transformations such as loop-nest optimization and parallelization on affine loop nests. Pluto transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. OpenMP parallel code for multicores can be automatically generated from sequential C program sections (Bondhugula, 2016). The polyhedral model representation is provided in Bondhugula *et al.* (2008). Bondhugula *et al.* (2008) have presented the design and implementation of a fully automatic polyhedral source-to-source program optimizer that can

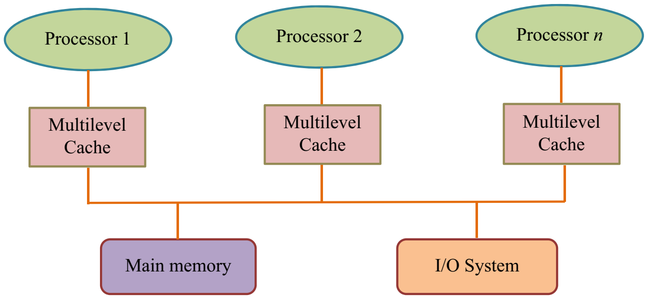


Figure 1. Shared memory architecture

simultaneously optimize sequences of arbitrarily nested loops for parallelism and locality. Through this work, they have shown the practicality and automatic transformation in the polyhedral model. They implemented the framework in Pluto to generate OpenMP parallel code from C program sections automatically.

Kallel *et al.* (2013) presented an empirical comparison between three research tools, namely, Cetus, Pluto and Gaspard (Devin *et al.*, 2002). They proved that the Pluto tool is more efficient than other two tools in terms of nested loop detection. Pluto tool gave optimal results for the benchmarks they considered. Pluto CLooG-ISL (Cloog.org, 2016) is used for code generation. CLooG (Chunky Loop Generator) is a free software and library to generate code for scanning Z-polyhedra. That is, it finds a code (e.g. in C, FORTRAN, etc.) that reaches each integral point of one or more parameterized polyhedra. CLooG has been originally written to solve the code generation problem for optimizing compilers based on the polytope model. PoCC (Verdoolaege *et al.*, 2013) is a full compiler for polyhedral optimization. It leverages most of the state-of-the-art polyhedral tools in the public domain, resulting in a flexible and powerful compiler. PoCC requires Perl to be installed in order to work properly. It also requires of course a Shell and a working C compiler. ROSE (Quinlan and Liao, 2011) is designed to support software analysis and optimization for both source code and binary software. As a source-to-source compiler infrastructure, it supports the automated analysis and transformation of large-scale applications as part of ongoing research to support compiler research, future computer architectures and software analysis and verification. S2P tool (Ranadive and Vaidya, 2011) is a fully automated parallelizing compiler that converts sequential C source code to a parallel multi-threaded source code for shared memory architecture. It performs data dependence analysis, which determines whether loop iterations operate on different array elements, to determine if the loop is parallelizable. It is a more contemporary tool as compared to the SUIF (The SUIF Compiler – Software Distribution, 2016) and Polaris but uses similar compiler optimization and parallelization techniques as the SUIF compiler.

2.2 The parallel programming frameworks

Following are the different parallel frameworks that are used for code parallelization.

2.2.1 OpenMP. OpenMP is a standard Application Programming Interface for writing shared memory parallel applications in C, C++ and FORTRAN. OpenMP is suitable for a shared memory parallel system. OpenMP includes a number of functions whose type must be declared in any program that uses them. The execution model of OpenMP is based on the fork/join model. An OpenMP begins as a single thread (called a master thread) and then defines parallel regions (by means of parallel directives) that are executed by multiple threads. Complex options allow for dividing loop iterations among threads, for defining shared and local variables in parallel regions and for defining reduction variables. Parallel regions are optimized by OpenMP compilers in a transparent way, as in shared memory management (Cantiello *et al.*, 2013; Computing.llnl.gov, 2016b).

The working of OpenMP is as follows:

- the parallel program is given as input to the processor with more than one core; and
- the outputs from each core are obtained and a single result from the whole is obtained as the output. Figure 2 shows the working of OpenMP.

2.2.2 MPI. MPI is a message passing library standard based on the consensus of the MPI forum, which has over 40 participating organizations, including vendors, researchers, software library developers and users (Vrenios, 2004; Kirk and Wen-Mei, 2016).

The working of MPI is as follows:

- The parallel program is given to the master node.

- The master node distributes the program to its *n* compute nodes.
- The result from each compute node is obtained and combined by the master to produce the final result. The working of MPI can be shown by just assuming the threads of Figure 2 as individual processors.

2.2.3 CUDA. CUDA is a scalable programming model and a software environment for parallel computing for NVIDIA GPUs. It exposes an SPMD programming model where the kernel is run by a large number of threads grouped into blocks. A typical program that uses GPUs has a host portion that runs on CPUs and device portion that are composed of kernels, which run on GPUs. The host portion controls the overall flow of the whole program and also controls the transfer of the data between GPUs and CPUs (Kirk and Wen-Mei, 2016). Processing flow of CUDA steps is as follows:

(1) copy processing data from main memory to memory for GPU;

(2) CPU will instruct the processor;

(3) GPU will execute parallel in each core; and

(4) copy the result from GPU memory to main memory.

3. Methodology

The methodology we proposed here should be able to scan the input file and produce the output for sequence code and parallel code for the specific classified problem used.

The main performance requirements are mentioned below:

- the system should be safe in case of large programs; it should not slow down the other operating processes running in the OS;
- the system must not go out of memory for large programs;
- the outputs from the sequential and parallel implementations should be the similar;
- the system should give correct output in a reasonable amount of time;
- while we parallelize, the overhead should not be increased; and
- in parallel implementation, it should provide better execution time comparative to the sequential one.

3.1 Design constraints and technical details

This section represents the important design decisions that have been made. The important decisions taken are to use OpenMP for hand parallelization and Pluto tool for auto parallelization.

The computer machine which is used for the experimentation is a notebook PC manufactured by HP (model Pavilion g6-1,213tx). It consist of 2.2 GHz Intel Pentium B960

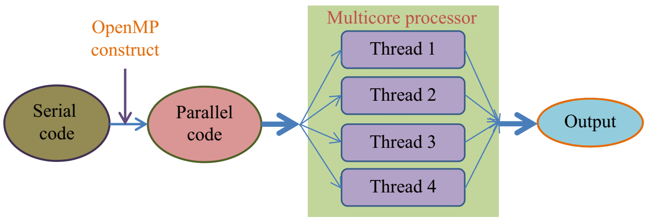


Figure 2. Working of OpenMP

processor with cache latency of 4 (L1 cache) 12 (L2 cache) 27 (L3 cache), 2 MB 8-way set associative shared cache L3 cache 4 GB 1,333 MHz DDR3 RAM, AMD Radeon HD 6,470M (1 GB DDR3 dedicated) video graphics card, 500 GB SATA (5,400 rpm) hard disk drive. The processor incorporates two CPU cores, operating at 2.2 GHz. Each core on this part has dedicated 64 KB L1 and 256 KB L2 caches, and both cores share 2 MB level 3 cache. The GCC compiler is used for the experimentations on Ubuntu 14.04 OS.

3.1.1 Overall working principle. We selected the specific algorithm like matrix multiplication, matrix vector multiplication, matrix addition, Floyd's algorithm and Dijkstra's algorithm from the numeric category; it is then translated to C code. Further, serialization as well as parallelization of the code is done later. Execution engine will execute the code and the results will be analyzed manually and the output will be a decision table.

Algorithms are selected based on the problems like numerical, non-numerical, affine, non-affine, regular and irregular from the computational domain. Algorithms such as matrix multiplication, matrix vector multiplication and matrix addition are numerical, affine, dense and regular. Floyd's algorithm and Dijkstra's algorithm are numerical, affine, dense and irregular. These algorithms are used as programs for our implementation.

The selected algorithms will be translated to C code. Serialization and parallelization of the code are done. Parallelization can be done in two ways, first way is to hand parallelize the code using different frameworks such as OpenMP, CUDA and MPI, and other way is to give the serial program with necessary modification as an input to the automatic parallel tool. The modification of the serial program will be based on the different automatic parallelization tool. These modifications include usage of pragmas to translate C code to OpenMP C. Automatic parallel tool will give us the parallel code.

3.1.2 Execution engine. Execution engine consists of compiler, debugger and shell scripts. Execution engine will execute the serial and parallel code. It is upon the execution engine to select the appropriate compiler and then set the size of the input and execute it. Execution engine makes use of GCC compiler for serial and hand parallel code. For automatic parallel tool, compiler and debugger may differ based on the configuration of automatic parallel tool.

3.1.3 Decision table. The results of the execution engine will be analyzed manually. After the manual analyzation, the output will be a decision table. Decision table consists of decisions which are later used in migration framework. Figure 3 explains the steps required for the proposed methodology.

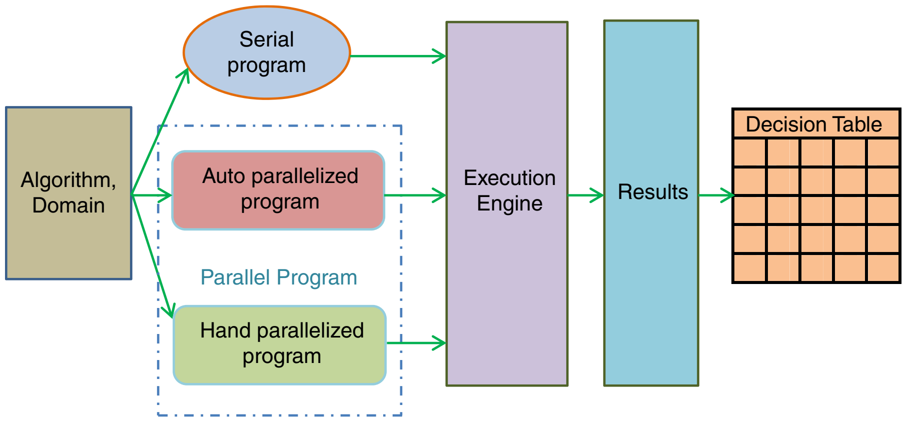


Figure 3.
Proposed methodology

Numerical problems are those which are used to solve mathematical problems and are measured in numbers. We have classified the types of the numerical algorithms based on classes such as affine, dense, regular and irregular. Affine problems are those where we can perform parallelization. A dense is a type of problem that contains a high percentage of occupied data values. Regular problems are those that have statically predictable behavior and irregular problems are those that have statically unpredictable behavior. Table I shows the classified problems based on numerical algorithms used in this work.

4. Execution and result analysis

The execution of the work is accomplished using the C language, OpenMP and Pluto-0.11.4 automatic parallelization tool on Ubuntu 14.04 operating system. The execution is performed with care by keeping only selected process running and suspending others. The experiments are conducted several times before we come to the conclusion. The implementation process is explained in the following steps.

For Sequential and OpenMP:

(1) select the algorithm and code it as serial and parallel program;

(2) compile the serial program; and

(3) run the program and note down the time taken for the execution.

For Pluto tool:

(1) enclose the serial code to be processed by Pluto within;

(2) compile the program; and

(3) run the program and note down the time taken for the execution.

Data sets were generated randomly for different data set sizes where the size of the data set was manually given. Following shows the execution time for the classified problems based on serialization, parallelization and auto parallelization of the algorithms.

4.1 Experimental results for matrix multiplication

The change in execution time for serial, hand parallel and auto parallelization keeping the number of rows *n* constant is observed. Table II shows the execution time of serial vs hand parallel vs auto parallel for matrix multiplication. Figure 4 shows the line chart of execution time of serial vs hand parallel and auto parallel code, where $n = 200$, 400, 600, 800, 1,000, 2,000, 3,000 and 4,000.

As the input size is increased, the difference between the total time taken to compute the final matrix with the serial code and the hand parallel code increases significantly. Figure 4 shows that automatic parallelization is efficient.

		Numeric			
S. No.	Types of problems	Dense	Affine	Regular	Irregular
1	Matrix multiplication	Yes	Yes	Yes	No
2	Matrix vector multiplication	Yes	Yes	Yes	No
3	Dijkstra's algorithm	Yes	Yes	No	Yes
4	Floyd's algorithm	Yes	Yes	No	Yes
5	Matrix addition	Yes	Yes	Yes	No

Table I. Problem classification

4.2 Experimental results for matrix vector multiplication

The change in execution time for serial, hand parallel and auto parallelization keeping n constant is observed. Table III shows the execution time of serial vs hand parallel vs auto parallel for matrix vector multiplication.

Figure 5 shows the line chart of execution time of serial vs hand parallel and auto parallel code, where $n = 2{,}000$, 4,000, 6,000, 8,000, 10,000, 12,000, 14,000, 16,000, 18,000 and 20,000. As the input size is increased, the difference between the total time taken to compute the final matrix with the serial code and the auto parallel code increases significantly. Figure 5 shows that hand parallelization is efficient for the matrix vector multiplication.

Table II. Runtime in matrix multiplication

Matrix input size N $(n\times n)\times(n\times n)$	Serial (in seconds)	Hand parallel (in seconds)	Auto parallel (in seconds)
200	0.0917	0.064012	0.034941
400	0.686434	0.43761	0.157935
600	2.893384	1.66029	0.4659
800	6.50812	4.244144	1.10877
1,000	16.18103973	7.65317425	2.136057
2,000	156.9904648	76.51662075	17.21628
3,000	556.9156003	294.701204	60.141463
4,000	1,276.824227	680.5741535	144.567

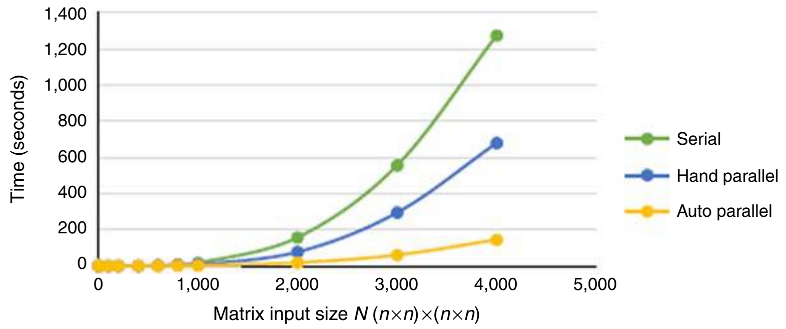


Figure 4. Execution time for matrix multiplication

Table III. Runtime in matrix vector multiplication

Matrix input size N $(n\times n)\times(n\times n)$	Serial (in seconds)	Hand parallel (in seconds)	Auto parallel (in seconds)
2,000	0.054	0.06187	0.067485
4,000	0.2048	0.14066	0.169075
6,000	0.4594	0.2939	0.342953
8,000	0.8169	0.52588	0.610054
10,000	1.2741	0.770337	0.93366
12,000	1.8324	1.0825	1.337037
14,000	2.474684	1.470105	1.916
16,000	3.22796	1.8956	2.479157
18,000	4.101386	2.376191	3.151066
20,000	5.068317	3.012502	3.862071

4.3 Experimental results for Dijkstra's algorithm

The change in execution time for serial, hand parallel and auto parallelization keeping n constant is observed. Table IV shows the execution time of serial vs hand parallel vs auto parallel for Dijkstra's algorithm. Figure 6 shows the bar chart of execution time of serial vs hand parallel and auto parallel code.

As the input size is increased, the difference between the total time taken to compute the minimum distance with the hand parallel code increases significantly. In Figure 6, it shows that auto parallelization and serialization give almost same execution time while running Dijkstra's algorithm.

4.4 Experimental results for Floyd's algorithm

The change in execution time for serial, hand parallel and auto parallelization keeping n constant is observed. Table V shows the execution time of serial vs hand parallel vs auto parallel. Figure 7 shows the line chart of execution time of serial vs hand parallel and auto parallel code, where $n = 10$, 100, 1,000.

As the input size is increased, the difference between the total time taken to compute the shortest path with the serial code and the hand parallel code increases significantly. In Figure 7, it shows that auto parallelization is efficient for Floyd's algorithm.

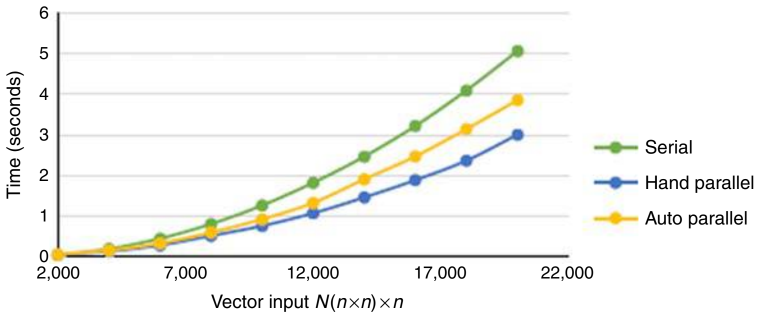
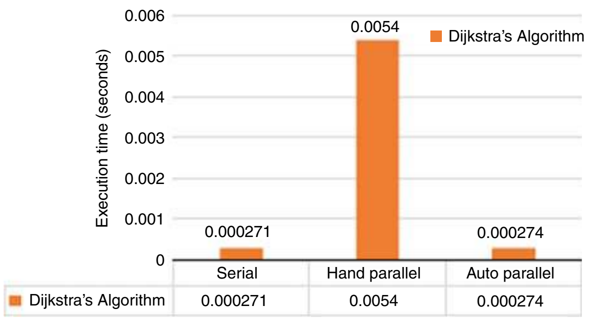


Figure 5. Execution time of matrix vector multiplication

Minimum distances	Serial (in seconds)	Hand parallel (in seconds)	Auto parallel (in seconds)
{0, 35, 15, 45, 49, 41}	0.00027	0.005438	0.00027

Table IV. Runtime in computing the minimum distances



	Serial	Hand parallel	Auto parallel
🟧 Dijkstra's Algorithm	0.000271	0.0054	0.000274

Figure 6. Execution time of Dijkstra's algorithm

4.5 Experimental results for matrix addition

The change in execution time for serial, hand parallel and auto parallelization keeping *n* constant is observed. Table VI shows the execution time of serial vs hand parallel vs auto parallel. Figure 8 shows the line chart of execution time of serial vs hand parallel and auto parallel code, where $n = 10$, 50, 100, 500, 1,000, 5,000 and 10,000.

As the input size is increased, the difference between the total time taken to compute the sum matrix with the serial code and the auto parallel code increases significantly. In Figure 8, it shows that hand parallelization is efficient for matrix addition.

Figure 9 shows the bar chart of execution time of serial, hand parallel and auto parallel code for classified algorithms where the least execution time is ranked as 20, moderate execution time is ranked as 40 and higher execution time is ranked as 60. Figure 9 clearly shows that parallelization is efficient than serialization. We analyzed that auto parallel tool Pluto is mainly designed to achieve the loop parallelization and the problems that we have chosen are strictly based on for loops. The only exception is the infinite loop, which may be written as either for (;;). The loop needs to be of the form for ($i = \text{init}(n)$; condition (n, i); $i += v$), where *n* is any number of parameters. Switch, break and continue statements are not supported by the current version of Pluto.

Table V.
Runtime in computing the Floyd's algorithm

Input size *n*	Serial (in seconds)	Hand parallel (in seconds)	Auto parallel (in seconds)
10	0.000037	0.000251	0.000009
100	0.010753	0.026629	0.002154
1,000	8.178548	19.63091	1.056928

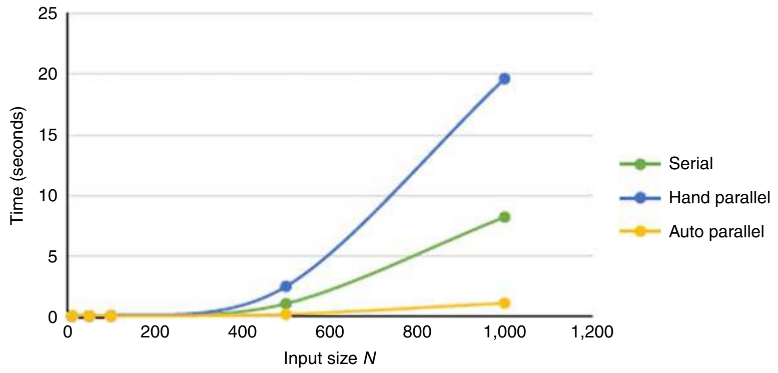


Figure 7.
Execution time of Floyd's algorithm

Table VI.
Runtime in computing the matrix addition

Input size N $(n \times n) + (n \times n)$	Serial (in seconds)	Hand parallel (in seconds)	Auto parallel (in seconds)
10	0.000012	0.006013	0.003422
50	0.000114	0.013373	0.019143
100	0.000492	0.01448	0.011726
500	0.011632	0.011125	0.004786
1,000	0.032234	0.023996	0.023104
5,000	0.572867	0.336016	0.472448
10,000	2.377163	1.239554	1.903838

After the analysis, we formed a decision table that a user can use while selecting the parallelization technique/framework for the classified problems. Table VII shows the decisions table. This decision tree provides the details that for which kind of classified problem which is the best (in terms of efficiency) technique to adopt.

5. Conclusion and future work

Auto parallelization tool is not matured enough to handle deskill to desk programs. Automatic parallelization tool is effective to certain type of problems and types of constructs. As we have observed that Pluto can parallelize numeric, dense, affine and regular problems, it cannot handle loop-carried dependencies. Main aspect of our project was to classify the problems based on numeric, dense, affine and regular. Based on the above classification, we have done serialization, hand parallelization and auto parallelization after selecting the tool as discussed. We analyzed that parallelization is better than serialization for the study. The classified programs will help the user to choose the suitable

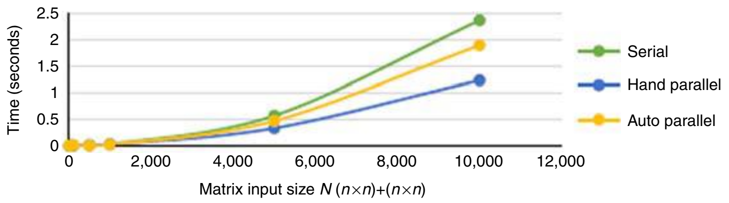


Figure 8. Execution time of matrix addition

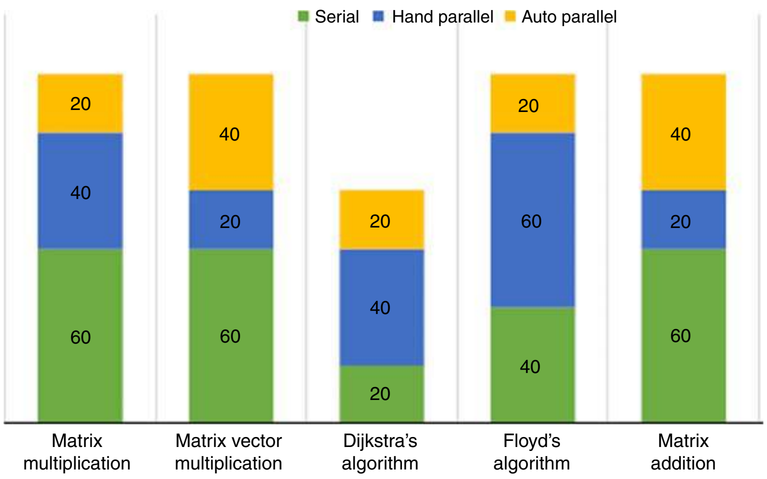


Figure 9. Execution time for all classified problems

Table VII. Decision table

S. No.	Programs	Serial execution	OpenMP execution	Pluto tool execution
1	Matrix multiplication			Best
2	Matrix vector multiplication		Best	
3	Dijkstra's algorithm			Best
4	Floyd's algorithm			Best
5	Matrix addition		Best	

approaches on shared memory architecture. Classified problems are successfully executed on the platform Linux and fed into automatic parallelization tool. The result is showed in tabular and graphical forms. Execution time for automatic parallelization tool and hand parallelization is noted down and bar graph is plotted. From the above study, we recommend users to choose matrix multiplication, Dijkstra's algorithm and Floyd's algorithm to parallelize using Pluto tool and matrix vector multiplication and matrix addition to be used in OpenMP. User can select the classified problem and establish the ease for migration of the application on the shared memory architecture using auto parallel tool and hand parallelization.

As Pluto is not able to handle loop-carried dependency, an advanced version Pluto+ has come up recently. This tries to solve the problem which Pluto could not handle. Pluto+ performs on heterogeneous architecture. One can analyze this tool on frameworks such as CUDA and MPI. These concepts can be used as the future extension of the parallelization work.

References

Al-Kharabsheh, K.S., AlTurani, I.M., AlTurani, A.M.I. and Zanoon, N.I. (2013), "Review on sorting algorithms a comparative study", *International Journal of Computer Science and Security*, Vol. 7 No. 3, pp. 120-127.

Athavale, A., Randive, P. and Kambale, A. (2011), "Automatic parallelization of sequential codes using s2p tool and benchmarking of the generated parallel codes", available at: www.kpit.com/downloads/research-papers/automatic-parallelization-sequential-codes.pdf (accessed January 15, 2017).

Bondhugula, U. (2016), "PLUTO – an automatic loop nest parallelizer and locality optimizer for multicores", available at: http://pluto-compiler.sourceforge.net/ (accessed May 5, 2016).

Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A. and Sadayappan, P. (2008), "A practical automatic polyhedral parallelizer and locality optimizer", *ACM SIGPLAN Notices*, Vol. 43 No. 6.

Cantiello, P., Di Martino, B. and Moscato, F. (2013), "Compilers, techniques, and tools for supporting programming heterogeneous many/multicore systems", *Large Scale Network-Centric Distributed Systems*, pp. 31-52.

Cloog.org (2016), "The Chunky Loop generator home", available at: www.CLooG.org (accessed May 5, 2016).

Computing.llnl.gov (2016b), "OpenMP", available at: https://computing.llnl.gov/tutorials/openMP/ (accessed January 15 , 2017).

Culler, D.E., Singh, J.P. and Gupta, A. (1999), *Parallel Computer Architecture: A Hardware/Software Approach*, Gulf Professional, Morgan Kaufmann Publishers, San Francisko, CA.

Dave, C. *et al.* (2009), "Cetus: a source-to-source compiler infrastructure for multicores", *Computer*, Vol. 42 No. 12, pp 36-42.

Devin, F. *et al.* (2002), "GASPARD: a visual parallel programming environment", *Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC'02)*, IEEE.

Gopan, D., Reps, T. and Sagiv, M. (2005), "A framework for numeric analysis of array operations", *ACM SIGPLAN Notices*, Vol. 40 No. 1, pp. 338-350.

Gupta, N. (2016), "Comparison and enhancement of sorting algorithms", *International Journal on Recent and Innovation Trends in Computing and Communication*, Vol. 4.2 No. 2, pp. 162-166.

Jindal, A., Jindal, N. and Sethia, D. (2012), "Automated tool to generate parallel CUDA code from a serial C code", *International Journal of Computer Applications*, Vol. 50 No. 8, pp. 15-21.

Kalender, M.E., Mergenci, C. and Ozturk, O. (2014), "AutopaR: an automatic parallelization tool for recursive calls", *2014 43rd International Conference on Parallel Processing Workshops (ICCPW)*, IEEE, September, pp. 159-165.

Kallel, E., Aoudni, Y. and Abid, M. (2013), "'OpenMP' automatic parallelization tools: an empirical comparative evaluation", *International Journal of Computer Science Issues (IJCSI)*, Vol. 10 No. 4, pp. 267-273.

Kirk, D.B. and Wen-Mei, W.H. (2016), "Programming massively parallel processors: a hands-on approach", Morgan kaufmann.

Kulalvaimozhi, V.P., Muthulakshmi, M., Mariselvi, R., Santhana Devi, G., Rajalakshmi, C. and Durai, C. (2015), "Performance analysis of sorting algorithm", *International Journal of Computer Science and Mobile Computing*, Vol. 4 No. 1, pp. 291-306.

Lattner, C. and Adve, V. (2004), "LLVM: a compilation framework for lifelong program analysis & transformation", *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, IEEE Computer Society*, pp. 75-87.

Nugteren, C. and Corporaal, H. (2012), "Introducing 'bones': a parallelizing source-to-source compiler based on algorithmic skeletons", *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, ACM, March*, pp. 1-10.

Parallel Processing & Parallel Databases (2016), "Parallel Processing & Parallel Databases", available at: https://docs.oracle.com/cd/A58617_01/server.804/a58238/ch1_unde.html (accessed April 28, 2016).

Pluto – Introduction (2016), "Pluto – Introduction", available at: www.cs.colostate.edu/~cs560/Spring2012/ClassNotes/Pluto_general_note.txt (accessed May 5, 2016).

Polaris (2016), "Polaris", available at: https://engineering.purdue.edu/paramnt/publications/polaris-nofront.pdf (accessed May 5, 2016).

Quinlan, D. and Liao, C. (2011), "The ROSE source-to-source compiler infrastructure", Cetus Users and Compiler Infrastructure Workshop, in Conjunction with PACT, Vol. 2011, October, p. 1.

Ranadive, P. and Vaidya, V.G. (2011), "Parallelization tool", available at: www.kpit.com/downloads/research-papers/parallelization-tools.pdf (accessed January 15, 2017).

Soleymani, F. (2012), "A rapid numerical algorithm to compute matrix inversion", *International Journal of Mathematics and Mathematical Sciences*, Vol. 2012, Hindawi Publishing Corporation, available at: http://dx.doi.org/10.1155/2012/134653

Thakkar, J., Parikh, M. and Panchal, B. (2017), "A parallel hybrid approach with MPI and OpenMP", *International Journal of Scientific Research in Science, Engineering and Technology*, Vol. 3 No. 6, pp. 584-588.

The SUIF Compiler – Software Distribution (2016), "The SUIF Compiler – Software Distribution", available at: http://suif.stanford.edu/suif/ (accessed May 5, 2016).

Thouti, K. and Sathe, S.R. (2013), "A methodology for translating C-programs to OpenCL", *International Journal of Computer Applications*, Vol. 82 No. 3, pp. 11-16.

Verdoolaege, S. *et al.* (2013), "Polyhedral parallel code generation for CUDA", *ACM Transactions on Architecture and Code Optimization*, Vol. 9 No. 4, pp. 2-24.

Vrenios, A. (2004), *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill.

Further reading

Computing.llnl.gov (2016a), "Message Passing Interface (MPI)", available at: https://computing.llnl.gov/tutorials/mpi/

Corresponding author
Tanvir Habib Sardar can be contacted at: tanvir.cs@pace.edu.in